External data representation

- https://developers.google.com/protocol-buffers/
- https://github.com/protobuf-c/protobuf-c

•
System data

- Internal data is represented as data structures
  - C structures/arrays
  - Java objects
- Transferred data is represented as byte sequences
- Data must be flatten to be transmitted
- Same data type can have multiple representations
  - e.g. floats/integers/characters
Data transmission

- Format of the transmitted data should be agreed
  - conversion to a common format
    - transmitter converts
    - receiver converts
  - transmitted in the sender format
    - receiver converts
Data transfer

- What kind of protocol to use, and what data to transmit?
- Efficient mechanism for storing and exchanging data

- Requirements
  - Correction
  - Efficiency
  - Interoperability (language/OS)
  - Ease to use
Data representation

- CORBA
  - Overdesigned and heavyweight
- Java object serialization
  - Tailored to one environment: Java
- DCOM, COM+
  - Tailored to one environment: Windows
- JSON, Plain Text, XML
  - Lack protocol description.
  - Programmer has to maintain both client and server code.
  - XML has high parsing overhead.
  - Relatively expensive to process; large due to repeated tags
## JSON vs XML

### Similarities:
- Both are human readable
- Both have very simple syntax
- Both are hierarchical
- Both are language independent

### Differences:
- Syntax is different
- JSON is less verbose
- JSON includes arrays
- Names in JSON must not be JavaScript reserved words
- XML can be validated
- JavaScript is not typically used on the server side

### Additional
- Still require explicit parsing and processing by the programmer
How to represent data?

• How to represent data?

• Which data types do you want to support?
  – Base types, Flat types, Complex types

• How to encode data into the wire

• How to decode the data?
  – Self-describing (tags)
  – Implicit description (the ends know)

• Several answers:
  – Many frameworks do these things automatically
Data Schema

- How to parse the encoded data?
- Two Extremes:
  - Self-describing data: tags
    - Additional information added to message to help in decoding
    - Examples: field name, type, length
  - Implicit: the code at both ends “knows” how to decode the message
    - Interoperability depends on well defined protocol specification!
    - Very difficult to change
Frameworks

- CORBA
  - Client
  - Local ORB
  - IIOP
  - Stubs
  - (de)serialization
  - DII
  - Services
  - Local ORB

- Generic Frameworks
  - IDL
  - Compiler
  - Stubs
    - (de)serialization
  - Supporting services
<methodCall>
  <methodName>
    sample.add
  </methodName>
  <params>
    <param>
      <value><int>17</int></value>
    </param>
    <param>
      <value><int>13</int></value>
    </param>
  </params>
</methodCall>

- XML is extremely verbose, which affects performance
- The library doesn’t support protocol versioning
  - What happens if I want to have another param?
- What happens if I reverse order of x and y?
  - In this case, nothing, but what if function weren’t commutative?
- What if I forget to add a param?
- In general, lack of types makes it difficult to build & maintain code
Stub Generation

• Many systems generate stub code from independent specification: IDL
  – IDL – Interface Description Language
    • describes an interface in a language neutral way

• Separates logical description of data from
  – Dispatching code
  – Marshalling/unmarshalling code
  – Data wire format
Apache Thrift

- Originally developed by Facebook
- Used heavily internally
- Full RPC system
- Support for C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and Ocaml
- Many types
  - Base types, list, set, map, exceptions
- Versioning support
- Many encodings (protocols) supported
- Efficient binary, json encodings
Typical Operation Model

- Write down a bunch of struct-like message formats in an IDL-like language.
- Run a tool to generate Java/C++/whatever boilerplate code.
  - Example:
    - `thrift --gen java MyProject.thrift`
- Outputs thousands of lines, but they remain fairly readable in most languages
- Link against this boilerplate when you build your application.
Google Protocol Buffers

- Defined by Google, released to the public
- Widely used internally and externally
- Supports common types, service definitions
- Natively generates C++/Java/Python code
  - Over 20 other supported by third parties
- Not a full RPC system, only does marshalling
  - Many third party RPC implementations
- Efficient binary encoding, readable text encoding
- Performance
  - 3 to 10 times smaller than XML
  - 20 to 100 times faster to process
Google Protocol Buffers

• Properties
  – Efficient, binary serialization
  – Support protocol evolution
    • Can add new parameters
    • Order in which I specify parameters is not important
    • Skip non-essential parameters
  – Supports types
    • which give you compile-time errors!
  – Supports somewhat complex structures

• Usage
  – Pattern: for each RPC call, define a new “message” type for its input and one for its output in a .proto file
  – Protocol buffers are used for other things, e.g., serializing data to non-relational databases
    • their backward compatible features make for nice long-term storage formats
  – Google uses them everywhere (50k proto buf definitions)
Google Protocol Buffers

1. .proto file
2. Protocol buffer compiler
3. => .java, .cc, .py
4. Client code
5. Client stub
6. Message classes
7. Server stub
8. Server code
9. javac, jar, gcc
10. Compiler/linker
11. Client-side program
12. Server-side program
13. RPC
14. 3rd-party library
Google Protocol Buffers

- Support service definitions and stub generation, but don't come with transport for RPC
  - There are third-party libraries for that
Protobuf vs. XML

- Protobufs are marshaled extremely efficiently
  - Binary format (as opposed to XML’s textual format)

- Example (according to protobuf documentation):

  ```
  <person>
    <name>John Doe</name>
    <email>jdoe@example.com
  </email>
  </person>

  person {
    1:“John Doe”
    3:”jdoe@example.com”
  }

- size:
  - 69 bytes (w/o whitespaces)

- parse:
  - 5,000-10,000ns
The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data such that:

- Serialization process is efficient, extensible and simple to use
- Serialized data can be stored or transmitted over the network

In Protocol buffers, Google has designed a language to specify messages
Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by
  - field-type,
  - Data-type
  - Field-name encoding-value
  - default value]
- Available data-types
  - Primitive data-type
  - int, float, bool, string, raw-bytes
  - Enumerated data-type
  - Nested Message
    - Allows structuring data into an hierarchy

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  repeated PhoneNumber phone = 4;
}
Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by
  - field-type,
  - Data-type
  - Field-name
  - encoding-value
  - default value
- Available data-types
  - Primitive data-type
    - int, float, bool, string, raw-bytes
  - Enumerated data-type
  - Nested Message
    • Allows structuring data into an hierarchy

```proto
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```
Protocol Buffer Language

- **Field-types can be:**
  - Required fields
  - Optional fields
  - Repeated fields
    - Dynamically sized array

- **Encoding-value**
  - A unique number
    - =1 =2 ...
  - represents a tag that a particular field has in the binary encoding of the message

```protobuf
define message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}
```
The specification of the message is contained in a .proto file

The .proto file is compiled by protoc tool

- The output of the protoc is a generated code that allows programmers to manipulate the particular message type

For example, assigning, extracting values to/from messages

```java
public static final class Person extends
boolean hasId();
int getId();
boolean hasEmail();
java.lang.String getEmail();
```

```proto
struct _Person{
  ...
  Person__PhoneNumber **phone;
};
```

Protoc-c

• Define the .proto file following the language syntax. Example
  
  ```
  message M1{
  - required string str = 1;
  - optional int32 i = 2;
  - }
  ```

• `protoc-c --c_out=. amessage.proto`
  
  - Generate .h and .c files from the command-line in your current working directory:
Protoc-c

• Write a C programm using
  – Include generated .h file
  – Generated structures
    • M1, Person, PhoneType, PhoneNumber
  – Generated functions
    • m1__init
    • m1__get_packed_size
    • m1__pack
    • m1__pack_to_buffer
    • m1__unpack
    • m1__free_unpacked
  – link with generate .c file libprotoc
Message Type

- message SearchRequest {
  - required string query = 1;
  - optional int32 page_number = 2;
  - optional int32 result_per_page = 3;
- }

Message name: SearchRequest

Fields:
- Field type
- data type
- Field name
- Field tag
Protoc-c

- Message
  - message MMM {
    - ...
    - }
- Generates the structure
- Generates the init function
  - MM__init
- Generates the pack functions (structure → buffer)
  - MMM__get_packed_size
  - MMM__pack
  - m1__pack_to_buffer
- Generates the unpack functions (buffer → structure)
  - m1__unpack (verifies if buffer contains a correct message
  - m1__free_unpacked
<table>
<thead>
<tr>
<th>Data type</th>
<th>Notes</th>
<th>C++ Type</th>
<th>Java Type</th>
<th>Python Type[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td></td>
<td>double</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>int32</td>
<td>Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>int64</td>
<td>Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.</td>
<td>int64</td>
<td>long</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>uint32</td>
<td>Uses variable-length encoding.</td>
<td>uint32</td>
<td>int[1]</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>uint64</td>
<td>Uses variable-length encoding.</td>
<td>uint64</td>
<td>long[1]</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>sint32</td>
<td>Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>sint64</td>
<td>Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.</td>
<td>int64</td>
<td>long</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>fixed32</td>
<td>Always four bytes. More efficient than uint32 if values are often greater than $2^{28}$.</td>
<td>uint32</td>
<td>int[1]</td>
<td>int</td>
</tr>
<tr>
<td>fixed64</td>
<td>Always eight bytes. More efficient than uint64 if values are often greater than $2^{56}$.</td>
<td>uint64</td>
<td>long[1]</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>sfixed32</td>
<td>Always four bytes.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>sfixed64</td>
<td>Always eight bytes.</td>
<td>int64</td>
<td>long</td>
<td>int/long[3]</td>
</tr>
<tr>
<td>bool</td>
<td></td>
<td>bool</td>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>string</td>
<td>A string must always contain UTF-8 encoded or 7-bit ASCII text.</td>
<td>string</td>
<td>String</td>
<td>str/unicode[4]</td>
</tr>
<tr>
<td>bytes</td>
<td>May contain any arbitrary sequence of bytes.</td>
<td>string</td>
<td>ByteString</td>
<td>str</td>
</tr>
</tbody>
</table>
Protoc-c data types

- Access fields in the message as in C structures
  - msg.a = 12

- Strings are defined as char *
  - Programmer must malloc memory for string in message
    - msg.str = malloc(.....)
  - Copy string to memory
    - strcpy(msg.st, ...)

Field Rules

• Required
  – a well-formed message must have exactly one of this field.

• Optional
  – a well-formed message can have zero or one of this field (but not more than one).

• Repeated
  – this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

• Required Is Forever
  – You should be very careful about marking fields as required.
  – If at some point you wish to stop writing or sending a required field,
    • It will be problematic to change the field to an optional field
    • old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally.
Optional fields

• The message definition can contain a default message field value:
  – optional int32 result_per_page = 3 [default = 10];

• If no default type is defined, a type-specific value is used:
  – for strings, the default value is the empty string
  – for bools, the default value is false
  – for numeric types, the default value is zero.
  – for enums, the default value is the first value listed in the enum's type definition.
Protoc-c optional fields

• message SearchRequest {
  •   optional int32 page_number = 2;
  •   ...
  • }

• The C structure will also contain
  – Int has_page_number (a boolean)
  – If set to true the value will be transmitted
Protoc-c repeated fields

• message CMessage {
  • repeated int32 array=1;
  • }

• The C structure will contain
  – int n_array
    • Integer containing the number of elements of array
  – int * array
    • Programmer should
      – msg.array = malloc(... * n_array)
      – Msg.array[i] = k;
Protoc-c repeated strings

- message CMessage {
  - repeated string list_strings=1;
} 

- The C structure will contain
  - int n_list_strings
    - Integer containing the number of elements of array
  - char ** list_strings
    - Programmer should
      - Create array of strings
      - Create each string
Enumerate fields

- Allows a field to have only have one of a pre-defined list of values.
- Add enum definition to the message:
  - message SearchRequest {
    - ...
    - enum Corpus {
      - UNIVERSAL = 0;
      - WEB = 1;
      - IMAGES = 2;
    }
  }
- Define a field:
  - optional Corpus corpus = 4 [default = UNIVERSAL];
  - }
message SearchRequest {
...

    enum Corpus {
        UNIVERSAL = 0;
        WEB = 1;
        IMAGES = 2;
    }

    optional Corpus corpus = 4 [default = UNIVERSAL];
}

typedef enum _M2__Corpus {
    M2__CORPUS__UNIVERSAL = 0,
    M2__CORPUS__WEB = 1,
    M2__CORPUS__IMAGES = 2,
} M2__Corpus;
Using Message Types

message SearchResponse {
    repeated Result result = 1;
}

message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
}

- Protoc-c
  - Creates message type
  - Creates fields
    - Pointer to structure
- New type
  - Programmed as C structures
- Optional
  - Pointer == NULL
References

- https://developers.google.com/protocol-buffers/
- https://developers.google.com/protocol-buffers/docs/proto
- https://developers.google.com/protocol-buffers/docs/docs/tutorials
- https://github.com/protobuf-c/protobuf-c
- https://github.com/protobuf-c/protobuf-c/wiki